Fall Semester, 2004 **B.Y. Choueiry**

CSCE 421/821: Foundations of Constraint Processing

Homework 2

(Updated at 1:00 pm)

Assigned: Wednesday, September 15, 2004

Due: Friday, September 24, 2004

Total value: 80 points

Notes: This homework must be done individually. *If you receive help from anyone, you must clearly acknowledge it.* Always acknowledge sources of information (URL, book, class notes, etc.). Please inform instructor quickly about typos or other errors.

Contents

1 Basic data structures **2**

1.1 Main data structures 2

1.2 Main functions/methods 4

2 Loading and initializing a CSP instance **4**

2.1 Constraints defined in extension 4

2.2 Constraints defined in intension 5

2.3 File format 6

2.4 Example file: 4-queen problem 8

2.5 Example file: 6-queen problem 8

2.6 Example file: Zebra 10

2.7 Example file: Graduation-picture puzzle 11

Implementation of a generic CSP

(Total 80 points)

The goal of this exercise is to write code for generating the data structures and accessor functions that will allow you to implement (most) CSPs, and to test your implementation by reading examples from files. This implementation may need to be refined as we progress in the course, but at least

the basic building-blocks for initializing, storing, and manipulating CSPs should be available. It is very important that you take this task seriously and do the implementation as clearly and neatly as possible: future homework will build upon this code. This is the first time we try to specify such data structures on paper, although most of us have repeatedly done such an implementation. Consequently, there may some errors in the design or the description: please quickly mention them to the instructor as you encounter them.

This exercise has two parts: one for creating the basic data structures (60 points), and the other one for generating the encoding of a CSP instance by reading the date from a file (20 points).

1 Basic data structures

Below we specify (as best we can) the data structures that need to be defined for storing the encoding of a CSP. When generating an encoding, we will generate instances of these general data structures. *We will restrict ourselves to binary CSPs. Those looking for a real challenge may want to consider generalizing their code, now or later, to non-binary CSPs.*

1.1 Main data structures

1. *all-problems.* Create a global variable (e.g., a linked list) for storing all CSP instances generated. Every time a CSP instance is generated, it should be pushed (preferably automatically) into this list. This is the data structure that we will go to in order to access any of the problems stored in memory.

2. *Problem instance.* Create a record-like data-structure for storing a CSP instance. (For lispers, use `defstruct`.) This data structure should have the following attributes:

   - `id`: that can be given or automatically generated (e.g., using a generator of strings).
   - `variables`: to be filled with a list of the data structures corresponding to the variables.
   - `constraints`: to be filled with a list of the data structures corresponding to the constraints.
   - `values`: a list of pointers to all the structures of values in the problem. Such a list is usually useful in scheduling and resource allocation applications to store all resources.

3. *Variable.* Create a record-like data-structure for storing a CSP variable. This data structure should have the following attributes:

   - `id`: that can be either given or automatically generated (e.g., using a generator of strings).
   - `problem`: a pointer to the problem data-structure to which the variable belongs.
   - `z-variable`: either a variable name (e.g., a string, an integer) or a pointer to a data-structure representing a variable.

- `initial-domain`: to be filled with a list of data structures of the values in the domain of the variable.
- `constraints`: to be filled with a list of the data structures corresponding to the constraints that apply to the variable.
- `neighbors`: to be filled with a list of the other CSP variables sharing a constraint with the variable.

4. *Value.* Create a record-like data-structure for storing a CSP value. This data structure should have the following attributes:

   - `id`: that can be either given or automatically generated (e.g., using a generator of strings).
   - `z-value`: either a value name (e.g., a string, an integer) or a pointer to a data-structure representing a value.

5. *Constraint.* Use a *class* to declare and store a CSP constraint. This data structure should have the following attributes:

   - `id`: that can be either given or automatically generated (e.g., using a generator of strings).
   - `problem`: a pointer to the problem data-structure where the constraint appears.
   - `variables`: a list of pointers to the variables in the scope of the constraint.
   - `definition`: to store the definition of a constraint: either as a set of tuples (for a constraint defined in extension) or as a function (for a constraint defined in intention).

6. *Extensional constraint.* Use a *class* to declare and store a CSP constraint declared in extension. This class should be a subclass of the constraint class, and thus inherit its attributes.

   The `definition` slot of the constraint should store the (linked) list of values that are allowed by the constraint in the lexicographical order of the variables. For instance:

   $$C_{V_1,V_2} = \{(1,2),(3,5),(2,3),\ldots\} \tag{1}$$

   Alternatively, you could store the allowed tuples in an array of two dimensional array of size $d^2 \times 2$, where d is the max domain-size (more generally, the size is $d^k \times 2$, where k is the max arity). Each row in this array is a tuple allowed by the constraint. (This is similar to a table in a relation database.)

7. *Intentional constraint.* Use a *class* to declare and store a CSP constraint declared in intention. This class should be a subclass of the constraint class, and thus inherit its attributes.

   The `definition` slot of the constraint should store a function that determines whether a list of variable-value pairs given in lexicographical order of the variables satisfies the constraint. For the time being, restrict yourself to variable-value pairs corresponding to the variables in the scope of the constraint. This is the function that `consistent-p` (see below) will need to access.

1.2 Main functions/methods

Further, create the following methods that apply to this constraint data-structure:

- `arity`: should return the number of variables in the scope of the constraint.
- `unary-p`: should return 1 if the constraint applies exactly to one variable and 0 otherwise.
- `binary-p`: should return 1 if the constraint applies exactly to two variables and 0 otherwise.
- `instantiated-vars`: should return the list of variables in the scope of the constraint that have been instantiated.
- `unassigned-vars`: should return the list of variables in the scope of the constraint that have not been instantiated.
- `consistent-p`: checks whether the instantiations of the instantiated variables in the scope of the constraint (i.e., the set of variable-value pairs) listed in sorted order of the instantiated variables are consistent with the constraint definition. The implementation of this method should take into account whether the constraint is declared as an intensive constraint or an extensive one.

  (Advanced note for non-binary constraints: careful for the arity of the tuple given as input as it may be different from that of the constraint. This requires some thinking...)

2 Loading and initializing a CSP instance

You are requested to write a parser that reads the description of a CSP as shown below and generates an encoding of it according to the data structures you have defined above. Below, we *tentatively* specify the format of the files you will have to read to load the definition of a CSP instance and generate the corresponding data structures, which you will then have to use to solve the problem with search. The content of this section is adapted from Catherine Anderson.

We strongly advise you to use `lex` and `yacc`, which will tremendously facilitate this task. It is worth the investment to learn how to use them.

The ability of your code to successfully read and correctly encode each CSP will be graded 5 points.

2.1 Constraints defined in extension

The constraint definition is given as a set of tuples after the names of the variables over which is it defined.

2.2 Constraints defined in intension

Below we provide some 'standard' names for constraints defined in intension. In the problem definition, the name will be preceded by the sign #'. Assuming P1 and P2 are the variable names that appear as the first two parameters of each constraint, consider implementing, *as you need them*, the following constraints:

- #'> : greater than (P1 > P2)
- #'+1 : incremented by 1 than (P2=P1+1)
- #'geq : greater than or equal to (P1 $\geq$ P2)
- #'< : less than (P1 < P2)
- #'leq : less than or equal to (P1 $\leq$ P2)
- #'mutex : not equal to (P1 $\neq$ P2)
- #'nextTo : P1= (P2+1) or (P2-1)
- #'beyond : P1 $\leq$ (P2+1,P2,P2-1)
- #'diagonal : |P1-P2| $\neq$ |i-j| (n-queen constraint)

2.3 File format

The text below gives the definition of the problem to be solved, and will be sent as a file to the class by email. This file is a command line argument for the CSP Parser. The structure of the data file is as follows:

```
First bracket - problem Identification (alphabet only,  not digits
or other characters).

Example Line:

{ name of problem}

second bracket - Variables, where # is the number of variables

Format of Line:

{ # { variableName1, variableName2, ..., variableNameN}}

third bracket - domain range - will determine the minimum and maximum
domain range and the increment between elements.  This first number
is the lowest domain element, the second number is the highest domain,
and the third number is the increment between element.  The example
given below will generate the following domain:  {1,4,7,10}

Example Line:

{1,10,3}

Fourth bracket - Unary constraints.  List only those that have them.
Give variable name and a listing of the values allowed.  If there are
no unary constraints, include the empty braces: { }.

Example Line:

{  {variableName2{4,7}}
    {variableName5{1,10}}    }

Fifth bracket - binary constraint. List only one direction. Parser
will take the intersection of the allowed tuples of multiple
constraints between the same two variables.

Example Line:
```

```
{ {variableName1, variableName4, #'=}
{variableName5, variableName13, #'+3}   }
```

Sixth bracket - Special case: Large mutex groups. To be used when the mutex is between more than two variables. The first number is the number of mutex groups. This is followed by the listing of all the variables in each group, enclosed in braces.

Example Line:

```
{2 {VariableName1,variableName2,variableName3,variableName4,variableName5}
  {VariableName6,variableName7,variableName8,variableName9,variableName10}   }
```

Seventh bracket - Indicates which solver engine will be used (BT, BJ, CBJ, BM, BMJ, BMCBJ, FC, FCBJ, FCCBJ), whether or not AC3 will be run (true) or not (false) as a presearch process and whether a single solution (S) or all solutions (A) are wanted. The sample given indicates that BT search (chronological backtracking) will be run, without AC3 presearch processing, stpping after one solution.

Example Line:

```
{BT, false, S}
```

2.4 Example file: 4-queen problem

The full data file for the 4-queen problem is given below, starting and stopping before and after the line of ########.

```
##########################################################
{ Four Queen Problem Configuration}
{4 {Q1,Q2,Q3,Q4}}
{1,4,1}
{ }
{ {Q1,Q2,{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}}
{Q1,Q3,{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}}
{Q1,Q4,{(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,2),(4,3)}}
{Q2,Q3,{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}}
{Q2,Q4,{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)}}
{Q3,Q4,{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}}}
{}
{BT,false,S}
#############################################################
```

2.5 Example file: 6-queen problem

The full data file for the 6-queen problem is given below, starting and stopping before and after the line of ########.

```
##########################################################
{ Six Queen Problem Configuration}
{6 {Q1,Q2,Q3,Q4,Q5,Q6}}
{1,6,1}
{ }
{ {Q1,Q2,#'diagonal}
{Q1,Q3,#'diagonal}
{Q1,Q4,#'diagonal}
{Q1,Q5,#'diagonal}
{Q1,Q6,#'diagonal}
{Q2,Q3,#'diagonal}
{Q2,Q4,#'diagonal}
{Q2,Q5,#'diagonal}
{Q2,Q6,#'diagonal}
{Q3,Q4,#'diagonal}
{Q3,Q5,#'diagonal}
{Q3,Q6,#'diagonal}
```

```
{Q4,Q5,#'diagonal}
{Q4,Q6,#'diagonal}
{Q5,Q6,#'diagonal}}
{1 {Q1,Q2,Q3,Q4,Q5,Q6}}
{BT,false,S}
##############################################################
```

2.6 Example file: Zebra

The full data file for the zebra problem is given below, starting and stopping before and after the line of ########.

```
#####################################################################
{Zebra Problem configuration}
{25{Norway,England,Japan,Spain,Ukrane,blue,red,green,yellow,ivory,coffee,
    tea,oj,milk,water,horse,snail,zebra,fox,dog,Chesterfield,Parliment,
    LuckyStripe,Kool,OldGold}
}
{1,5,1}
{ {Norway{1}}
{milk{3}}
}
{ {England,red,#'=}
{Spain,dog,#'=}
{coffee,green,#'=}
{Ukrane,tea,#'=}
{snail,OldGold,#'=}
{Kool,yellow,#'=}
{LuckyStripe,oj,#'=}
{Japan,Parliment,#'=}
{green,ivory,#'+1}
{Chesterfield,fox,#'nextTo}
{Kool,horse,#'nextTo}
{Norway,blue,#'nextTo}
}
{5 {Norway,England,Japan,Spain,Ukrane}
{blue,red,green,yellow,ivory}
{coffee,tea,oj,milk,water}
{horse,snail,zebra,fox,dog}
{Chesterfield,Parliment,LuckyStripe,Kool,OldGold}
}
{BJ,false,S}
#########################################################################
```

2.7 Example file: Graduation-picture puzzle

An additional problem is given below: Graduation Pictures. The following puzzle is a typical logic problem found in logic puzzle books. There is a graduation at the local high school and a photographer takes a picture of Fay and 5 other young women (including Ms. Owens) in their graduation robes (each of a different color, including lavender) each standing on a step one higher then the person in front of them. From the following bits of information, find out the full names of the women, what color robes they each wore and which step they were standing on.

- Amy was on the step just below the Ms. Mertz and just above the women in green.
- Although Beth was not on the top step, she was above both the women in red and the Ms. Pinot.
- Coral stood below the Ms. Nash and above the women in green.
- Dana stood just below the women in blue and just above Ms. Lyons.
- Ms. Kelly stood just below Erin and just above the women in yellow.
- Ms. Mertz wasn't on step 4 and the Ms. Pinot wasn't on step 5.
- Coral did not wear yellow or pink.

```
Data file:
#################################################################
{Graduation Picture Problem Configuration}
{18 {Amy,Beth,Carol,Dana,Erin,Fay,blue,green,lavender,pink,red,
 yellow,Kelly,Lyons,Mertz,Nash,Owens,Pinot}}
{1,6,1}
{ {Mertz{1,2,3,5,6}}
{Pinot{1,2,3,4,6}}
{Beth{2,3,4,5,6}}
{Owens{2,3,4,5,6}}
}
{ {blue,Dana,#'+1}
{Dana,Lyons,#'+1}
{Erin,Kelly,#'+1}
{Kelly,yellow,#'+1}
{Mertz,Amy,#'+1}
{Amy,green,#'+1}
{red,Pinot,#'mutex}
{Carol,yellow,#'mutex}
{Carol,pink,#'mutex}
```

```
{red,Beth,#'>}
{Pinot,Beth,#'>}
{Carol,Nash,#'>}
{green,Carol,#'>}
}
{3 {Amy,Fay,Beth,Carol,Dana,Erin}
    {lavender,green,red,blue,yellow,pink}
{Owens,Mertz,Pinot,Nash,Lyons,Kelly}
}
{BJ,false,S}
##################################################################
```